# Exhibit 4

# U.S. Patent No. 7,784,058 vs. HPE

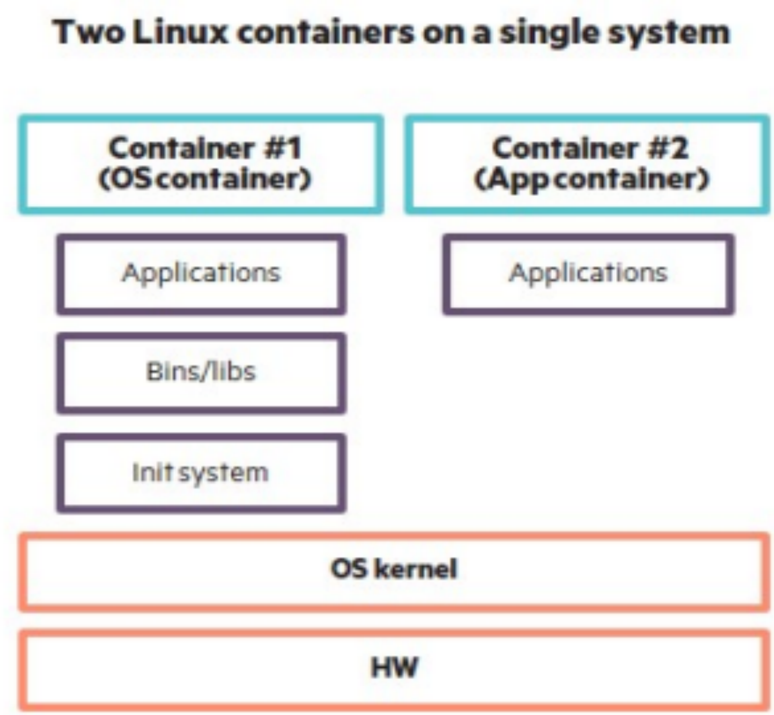
VirtaMove asserts that Defendant Hewlett Packard Enterprise Company ("Defendant" or "HPE") infringes the following claims (collectively, "Asserted Claims"): U.S. Patent No. 7,784,058 ("the '058 patent"), claims 1–4 and 18.

Accused Instrumentalities: HPE products and services using user mode critical system elements as shared libraries, including without limitation HPE's Ezmeral Runtime Enterprise (including without limitation both Ezmeral Runtime Enterprise and Ezmeral Runtime Enterprise Essentials, in each case including when marketed, sold, and/or licensed as part of or associated with HPE's GreenLake branding, e.g. "HPE GreenLake for containers" which "is built on HPE Ezmeral Container Platform"), and all versions and variations thereof since the issuance of the asserted patent.

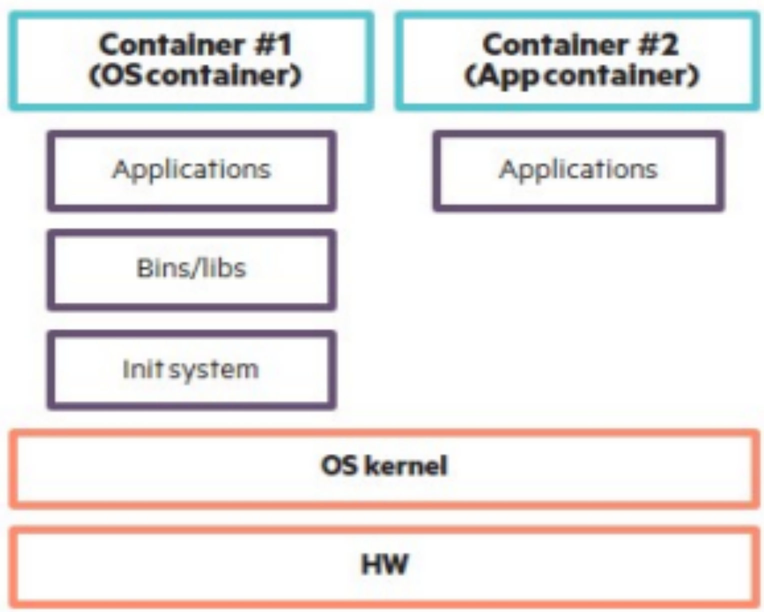
Each Accused Instrumentality infringes the claims in substantially the same way, and the evidence shown in this chart is similarly applicable to each Accused Instrumentality. Each claim limitation is literally infringed by each Accused Instrumentality. However, to the extent any claim limitation is not met literally, it is nonetheless met under the doctrine of equivalents because the differences between the claim limitation and each Accused Instrumentality would be insubstantial, and each Accused Instrumentality performs substantially the same function, in substantially the same way, to achieve the same result as the claimed invention. Notably, Defendant has not yet articulated which, if any, particular claim limitations it believes are not met by the Accused Instrumentalities.

## Claim 1

| Claim 1 | Accused Instrumentalities |
|---|---|
| [1pre] 1. A computing system for executing a plurality of software applications comprising: | To the extent the preamble is limiting, each Accused Instrumentality comprises or constitutes a computing system for executing a plurality of software applications as claimed.<br><br>*See* claim limitations below.<br><br>*See also, e.g.*:<br><br>**HPE Ezmeral Runtime Enterprise** is a unified platform built on open-source Kubernetes and designed for both cloud-native applications and non-cloud-native applications running on any infrastructure; whether on-premises, in multiple public clouds, in a hybrid model, or at the edge.<br><br>https://support.hpe.com/hpesc/public/docDisplay?docId=a00ecp54hen_us&page=home/about-hpe-ezmeral-container-pl/Welcome.html |

| Claim 1 | Accused Instrumentalities |
| --- | --- |
| | Containers provide the core runtime abstraction for the user applications. These containers provide isolation between user applications and the rest of the infrastructure. The containers are based on Docker.<br>https://support.hpe.com/hpesc/public/docDisplay?docId=a00097165en_us&docLocale=en_US&page=GUID-6B6676DB-AF5F-4555-B6AB-D2C11A89F320.html<br><br>https://h50146.www5.hpe.com/products/software/oe/linux/mainstream/support/whitepaper/pdfs/4AA6-2761ENW.pdf |
| [1a] a) a processor; | Each Accused Instrumentality comprises a processor.<br>For example, each node/host contains at least one CPU.<br>*See, e.g.*: |

| Claim 1 | Accused Instrumentalities |
| --- | --- |
| | Each license allows the customer to deploy the HPE Ezmeral Container Platform on one Core and 2 terabytes of Storage Capacity. The customer must purchase more licenses if they exceed the allowable amount of Cores or Storage Capacity. As used in this Agreement, Core means a part of a CPU that executes a single stream of compiled instruction code. Each physical processor contains smaller processing units called physical CPU cores. Some processors have two cores, some<br>https://support.hpe.com/hpesc/public/docDisplay?docId=a00ecp54hen_us&docLocale=en_US&page=home/about-hpe-ezmeral-container-pl/GEN_End_User_Software_Agreement.html |
| [1b] b) an operating system having an operating system kernel having OS critical system elements (OSCSEs) for running in kernel mode using said processor; and, | Each Accused Instrumentality comprises an operating system having an operating system kernel having OS critical system elements (OSCSEs) for running in kernel mode using said processor.<br>For example, the OSCSEs include kernel-mode functions similar to the functionalities provided by user-space libraries such as glibc. These are implemented in kernel-space to handle tasks such as (without limitation) memory management (kmalloc(), kfree(), etc,) at kernel level.<br>*See, e.g.*:<br>Containers provide the core runtime abstraction for the user applications. These containers provide isolation between user applications and the rest of the infrastructure. The containers are based on Docker.<br>https://support.hpe.com/hpesc/public/docDisplay?docId=a00097165en_us&docLocale=en_US&page=GUID-6B6676DB-AF5F-4555-B6AB-D2C11A89F320.html<br>Each license allows the customer to deploy the HPE Ezmeral Container Platform on one Core and 2 terabytes of Storage Capacity. The customer must purchase more licenses if they exceed the allowable amount of Cores or Storage Capacity. As used in this Agreement, Core means a part of a CPU that executes a single stream of compiled instruction code. Each physical processor contains smaller processing units called physical CPU cores. Some processors have two cores, some<br>https://support.hpe.com/hpesc/public/docDisplay?docId=a00ecp54hen_us&docLocale=en_US&page=home/about-hpe-ezmeral-container-pl/GEN_End_User_Software_Agreement.html |

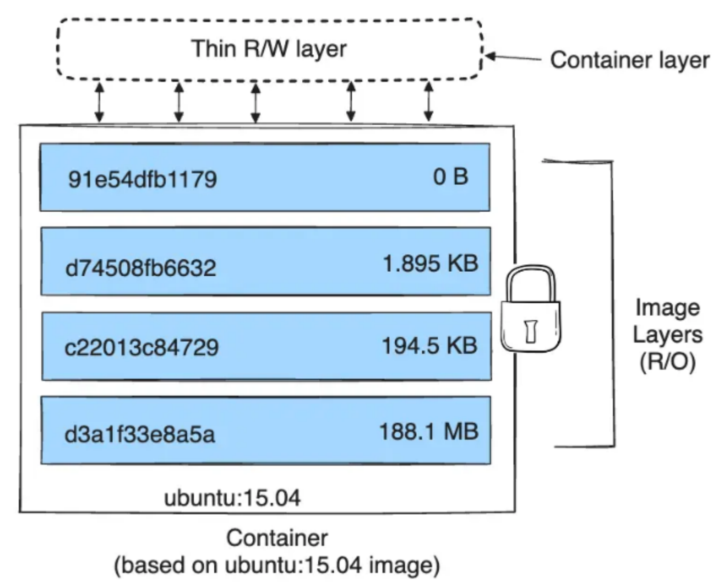
| Claim 1 | Accused Instrumentalities |
| --- | --- |
| | HPE Ezmeral Runtime Enterprise supports the following operating systems:<br><br>**HPE Ezmeral Runtime Enterprise Version** \| **CentOS Support** \| **RHEL Support** \| **SUSE Support**<br><br>https://support.hpe.com/hpesc/public/docDisplay?docId=a00ecp54hen_us&page=home/about-hpe-ezmeral-container-pl/GEN_OS_Support.html<br><br>**Two Linux containers on a single system**<br><br><br><br><br>https://h50146.www5.hpe.com/products/software/oe/linux/mainstream/support/whitepaper/pdfs/4AA6-2761ENW.pdf |

| Claim 1 | Accused Instrumentalities |
|---|---|
| | **Kernel mode**<br><br>Kernel mode refers to the processor mode that enables software to have full and unrestricted access to the system and its resources. The OS kernel and kernel drivers, such as the file system driver, are loaded into protected memory space and operate in this highly privileged kernel mode.<br><br>https://www.techtarget.com/searchdatacenter/definition/kernel<br><br>The **GNU C Library**, commonly known as **glibc**, is the GNU Project implementation of the C standard library. It is a wrapper around the system calls of the Linux kernel for application use. Despite its name, it now also directly supports C++ (and, indirectly, other programming languages). It was started in the 1980s by the Free Software Foundation (FSF) for the GNU operating system.<br><br>https://en.wikipedia.org/wiki/Glibc |
| [1c] c) a shared library having shared library critical system elements (SLCSEs) stored therein for use by the plurality of software applications in user mode and | Each Accused Instrumentality comprises a shared library having shared library critical system elements (SLCSEs) stored therein for use by the plurality of software applications in user mode.<br><br>For example, the shared library with SLCSEs include the runtime environment, system tools, and dependencies, such as the glibc library and other libraries that replicate OSCSEs, included in the container image (including without limitation in a base image that is included within the container image).<br><br>*See, e.g.*:<br><br>Containers provide the core runtime abstraction for the user applications. These containers provide isolation between user applications and the rest of the infrastructure. The containers are based on Docker.<br><br>https://support.hpe.com/hpesc/public/docDisplay?docId=a00097165en_us&docLocale=en_US&page=GUID-6B6676DB-AF5F-4555-B6AB-D2C11A89F320.html<br><br>The container starts with a base image, and the microservice is packaged into a container image and then deployed through the container platform. The container platform is based on<br><br>https://www.hpe.com/us/en/what-is/container-platform.html |

| Claim 1 | Accused Instrumentalities |
|---|---|
| | **Container images**<br><br>A container image is a ready-to-run software package containing everything needed to run an application: the code and any runtime it requires, application and system libraries, and default values for any essential settings.<br><br>https://kubernetes.io/docs/concepts/containers/<br><br>The idea of containerization is to isolate and package the application with all the dependencies in a container.<br><br>https://community.hpe.com/t5/hpe-blog-uk-ireland-middle-east/containerization-the-next-generation-of-virtualization/ba-p/7154442<br><br>Container image files are complete, static and executable versions of an application or service and differ from one technology to another. Docker images are made up of multiple layers, which start with a base image that includes all of the dependencies needed to execute code in a container. Each image has a readable/writable layer on top of static unchanging layers. Because each container has its own specific container layer that customizes that specific container, underlying image layers can be saved and reused in multiple containers. An Open Container Initiative (OCI)<br><br>https://www.techtarget.com/searchitoperations/definition/container-containerization-or-container-based-virtualization |

| Claim 1 | Accused Instrumentalities |
|---|---|
| | # About storage drivers<br><br>To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way.<br><br>## Storage drivers versus Docker volumes<br><br>Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer.<br><br>Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance.<br><br>https://docs.docker.com/storage/storagedriver/ |

| Claim 1 | Accused Instrumentalities |
| --- | --- |
| | # Images and layers<br><br>A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:<br><br>`# syntax=docker/dockerfile:1`<br><br>`FROM ubuntu:22.04`<br>`LABEL org.opencontainers.image.authors="org@example.com"`<br>`COPY . /app`<br>`RUN make /app`<br>`RUN rm -r $HOME/.cache`<br>`CMD python /app/app.py`<br><br>This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer.<br><br>https://docs.docker.com/storage/storagedriver/ |

| Claim 1 | Accused Instrumentalities |
|---|---|
| | Each layer is only a set of differences from the layer before it. Note that both *adding*, and *removing* files will result in a new layer. In the example above, the `$HOME/.cache` directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images.<br><br>The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image.<br><br><br><br><br>https://docs.docker.com/storage/storagedriver/ |

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

# Volumes

Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts:

https://kubernetes.io/docs/concepts/storage/volumes/

## Container environment

The Kubernetes Container environment provides several important resources to Containers:

- A filesystem, which is a combination of an image and one or more volumes.
- Information about the Container itself.
- Information about other objects in the cluster.

https://kubernetes.io/docs/concepts/containers/container-environment/

</td></tr>
</table>

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

# Images

A container image represents binary data that encapsulates an application and all its software dependencies. Container images are executable software bundles that can run standalone and that make very well defined assumptions about their runtime environment.

You typically create a container image of your application and push it to a registry before referring to it in a Pod.

https://kubernetes.io/docs/concepts/containers/images/

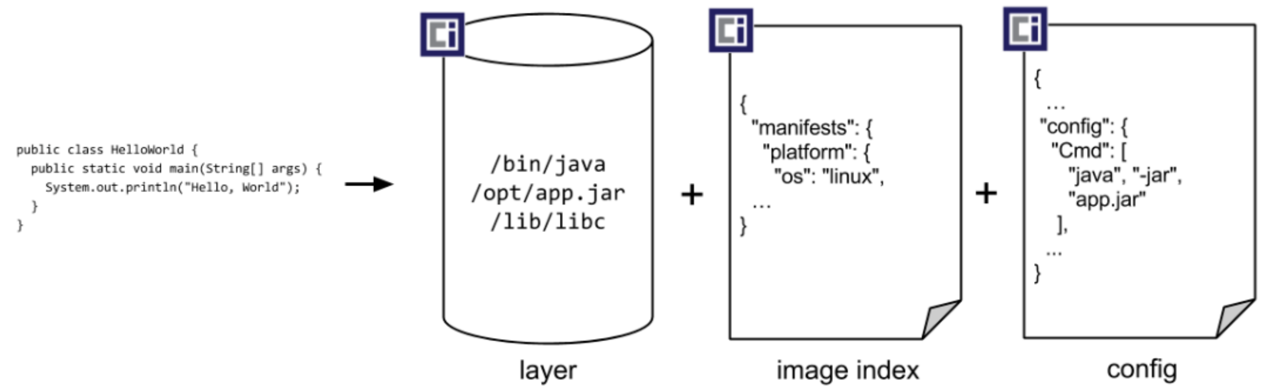
# Volumes

On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem occurs when a container crashes or is stopped. Container state is not saved so all of the files that were created or modified during the lifetime of the container are lost. During a crash, kubelet restarts the container with a clean state. Another problem occurs when multiple containers are running in a `Pod` and need to share files. It can be challenging to setup and access a shared filesystem across all of the containers. The Kubernetes volume abstraction solves both of these problems. Familiarity with Pods is suggested.

https://kubernetes.io/docs/concepts/storage/volumes/

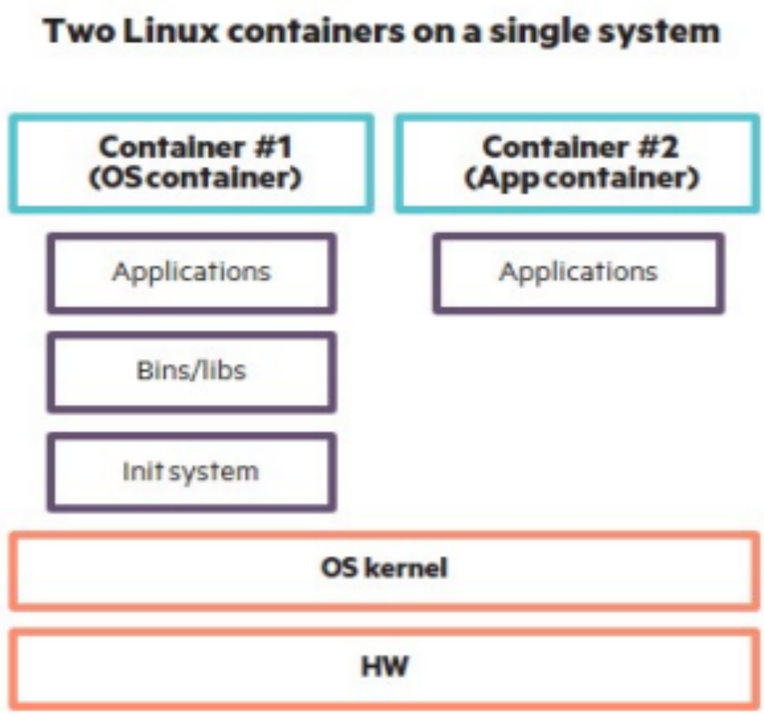
</td></tr>
</table>

| Claim 1 | Accused Instrumentalities |
|---|---|
| | **Open Container Initiative**<br><br>**Image Format Specification**<br><br>This specification defines an OCI Image, consisting of an image manifest, an image index (optional), a set of filesystem layers, and a configuration.<br><br>The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run.<br><br>https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md |

| Claim 1 | Accused Instrumentalities |
|---|---|
| | **Overview**<br><br>At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem layer changeset archives that will be unpacked to make up the final runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The image index is a higher-level manifest which points to a list of manifests and descriptors. Typically, these manifests may provide different implementations of the image, possibly varying by platform or other attributes.<br><br><br><br><br>https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md |

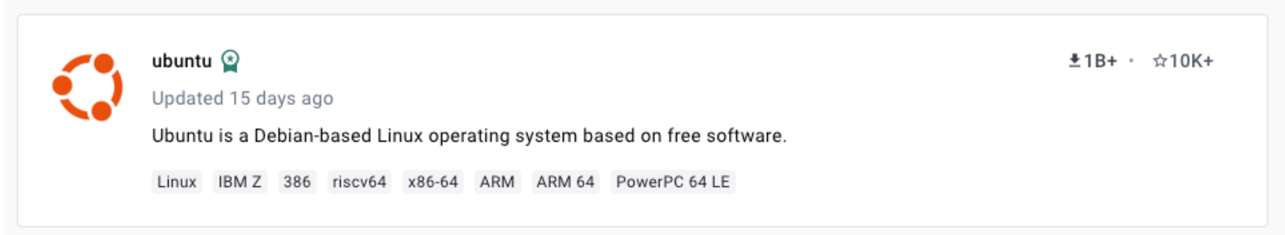
| Claim 1 | Accused Instrumentalities |
|---|---|
| | **OCI Image Configuration**<br><br>An OCI *Image* is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. This specification outlines the JSON format describing images for use with a container runtime and execution tool and its relationship to filesystem changesets, described in Layers.<br><br>This section defines the `application/vnd.oci.image.config.v1+json` media type.<br><br>https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md |

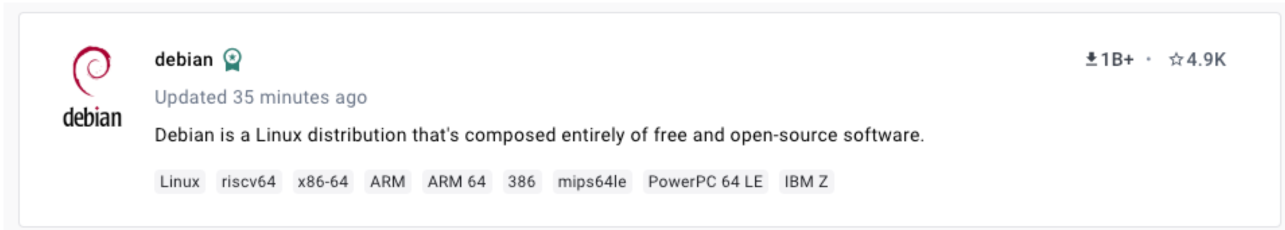
| Claim 1 | Accused Instrumentalities |
|---|---|
| | **Layer**<br>• Image filesystems are composed of *layers*.<br>• Each layer represents a set of filesystem changes in a tar-based layer format, recording files to be added, changed, or deleted relative to its parent layer.<br>• Layers do not have configuration metadata such as environment variables or default arguments - these are properties of the image as a whole rather than any particular layer.<br>• Using a layer-based or union filesystem such as AUFS, or by computing the diff from filesystem snapshots, the filesystem changeset can be used to present a series of image layers as if they were one cohesive filesystem.<br><br>**Image JSON**<br>• Each image has an associated JSON structure which describes some basic information about the image such as date created, author, as well as execution/runtime configuration like its entrypoint, default arguments, networking, and volumes.<br>• The JSON structure also references a cryptographic hash of each layer used by the image, and provides history information for those layers.<br>• This JSON is considered to be immutable, because changing it would change the computed ImageID.<br>• Changing it means creating a new derived image, instead of changing the existing image.<br><br>https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md |

| Claim 1 | Accused Instrumentalities |
|---|---|
| | <br>https://h50146.www5.hpe.com/products/software/oe/linux/mainstream/support/whitepaper/pdfs/4AA6-2761ENW.pdf<br><br>The **GNU C Library**, commonly known as **glibc**, is the GNU Project implementation of the C standard library. It is a wrapper around the system calls of the Linux kernel for application use. Despite its name, it now also directly supports C++ (and, indirectly, other programming languages). It was started in the 1980s by the Free Software Foundation (FSF) for the GNU operating system.<br><br>https://en.wikipedia.org/wiki/Glibc |
| [1d] i) wherein some of the SLCSEs stored in the shared library are functional replicas of OSCSEs and are accessible to some of the plurality of software applications and when one of the | In each Accused Instrumentality, some of the SLCSEs stored in the shared library are functional replicas of OSCSEs and are accessible to some of the plurality of software applications and when one of the SLCSEs is accessed by one or more of the plurality of software applications it forms a part of the one or more of the plurality of software applications. |

| Claim 1 | Accused Instrumentalities |
|---|---|
| SLCSEs is accessed by one or more of the plurality of software applications it forms a part of the one or more of the plurality of software applications, | For example, a base image serves as a self-contained unit that encompasses all the necessary components for an application to run, including the application code, runtime environment, system tools, and dependencies (i.e., SLCSEs). The images are based on existing Linux distributions, such as Debian and Ubuntu, including essential system elements (i.e., functional replicas of OSCSEs). Each container image is based on a specific base image, which contains the application code, and dependencies, including system libraries or shared library critical system elements (SLCSEs). The base image forms a part of the container image according to the "layer" model described in the documentation below. When the container runs the image, it creates a runtime instance of that container image. In turn, when one or more applications executes within the container runtime environment, it dynamically links to the SLCSEs stored in the runtime environment, which thereby become a part of the application(s).<br><br>*See, e.g.*:<br><br>Hewlett Packard Enterprise provides publicly available base OS images for use in containerized clusters. These images extend the base OS images available from Docker hub by adding several packages that permit HPE Ezmeral Runtime Enterprise to manage container orchestration seamlessly and to improve the security of the container.<br>https://docs.ezmeral.hpe.com/runtime-enterprise/55/app-workbench-5-1/custom-base-images/AWB51_About_Custom_Base_Images.html<br><br>The idea of containerization is to isolate and package the application with all the dependencies in a container.<br>https://community.hpe.com/t5/hpe-blog-uk-ireland-middle-east/containerization-the-next-generation-of-virtualization/ba-p/7154442<br><br>**Container images**<br><br>A container image is a ready-to-run software package containing everything needed to run an application: the code and any runtime it requires, application and system libraries, and default values for any essential settings.<br><br>https://kubernetes.io/docs/concepts/containers/ |

| Claim 1 | Accused Instrumentalities |
|---|---|
| | |



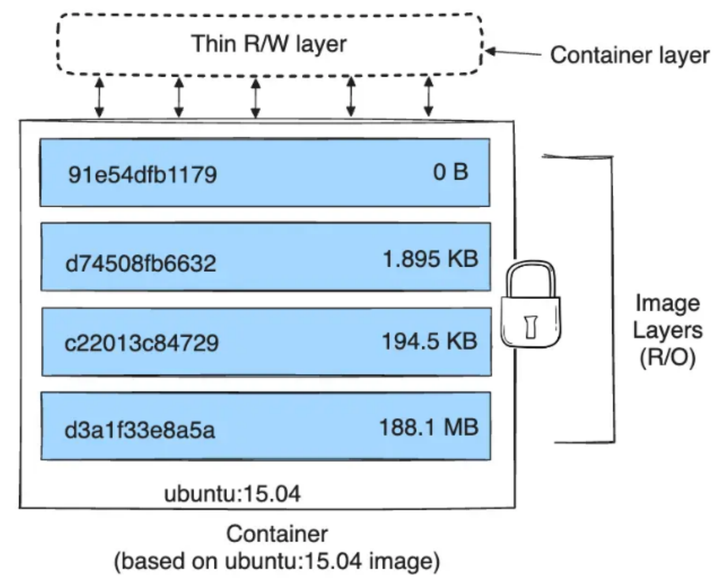

https://hub.docker.com/search?image_filter=official&type=image&q=

| Platform | x86_64 / amd64 | arm64 / aarch64 | arm (32-bit) | ppc64le | s390x |
|---|---|---|---|---|---|
| CentOS | ✅ | ✅ | | ✅ | |
| Debian | ✅ | ✅ | ✅ | ✅ | |
| Fedora | ✅ | ✅ | | ✅ | |
| Raspberry Pi OS (32-bit) | | | ✅ | | |
| RHEL (s390x) | | | | | ✅ |
| SLES | | | | | ✅ |
| Ubuntu | ✅ | ✅ | ✅ | ✅ | ✅ |
| Binaries | ✅ | ✅ | ✅ | | |

| Claim 1 | Accused Instrumentalities |
|---|---|
| | https://docs.docker.com/engine/install/<br><br>Docker is used to create, run and deploy applications in containers. A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container.<br><br>https://www.techtarget.com/searchitoperations/definition/Docker-image |

| Claim 1 | Accused Instrumentalities |
|---|---|
| | **About storage drivers**<br><br>To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way.<br><br>**Storage drivers versus Docker volumes**<br><br>Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer.<br><br>Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance.<br><br>https://docs.docker.com/storage/storagedriver/ |

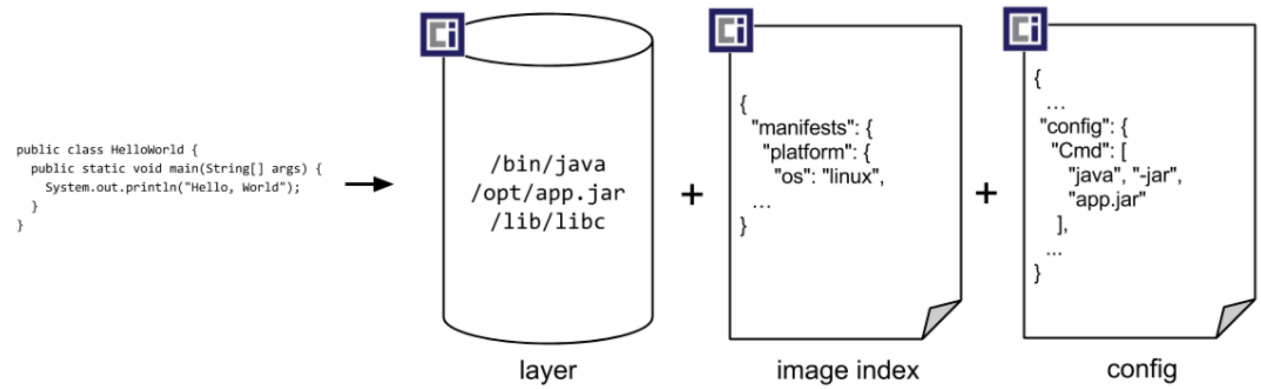
| Claim 1 | Accused Instrumentalities |
|---|---|
| | ## Images and layers<br><br>A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:<br><br>`# syntax=docker/dockerfile:1`<br><br>`FROM ubuntu:22.04`<br>`LABEL org.opencontainers.image.authors="org@example.com"`<br>`COPY . /app`<br>`RUN make /app`<br>`RUN rm -r $HOME/.cache`<br>`CMD python /app/app.py`<br><br>This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer.<br><br>https://docs.docker.com/storage/storagedriver/ |

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>
Each layer is only a set of differences from the layer before it. Note that both adding, and removing files will result in a new layer. In the example above, the <code>$HOME/.cache</code> directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images.<br><br>
The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an <code>ubuntu:15.04</code> image.<br><br>

<br>
https://docs.docker.com/storage/storagedriver/
</td></tr>
</table>

| Claim 1 | Accused Instrumentalities |
| --- | --- |
| | **Volumes**<br><br>Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts:<br><br>https://kubernetes.io/docs/concepts/storage/volumes/<br><br>**Container environment**<br><br>The Kubernetes Container environment provides several important resources to Containers:<br><br>• A filesystem, which is a combination of an image and one or more volumes.<br>• Information about the Container itself.<br>• Information about other objects in the cluster.<br><br>https://kubernetes.io/docs/concepts/containers/container-environment/ |

| Claim 1 | Accused Instrumentalities |
| --- | --- |
| | **Images**<br><br>A container image represents binary data that encapsulates an application and all its software dependencies. Container images are executable software bundles that can run standalone and that make very well defined assumptions about their runtime environment.<br><br>You typically create a container image of your application and push it to a registry before referring to it in a Pod.<br><br>https://kubernetes.io/docs/concepts/containers/images/<br><br>**Volumes**<br><br>On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem occurs when a container crashes or is stopped. Container state is not saved so all of the files that were created or modified during the lifetime of the container are lost. During a crash, kubelet restarts the container with a clean state. Another problem occurs when multiple containers are running in a `Pod` and need to share files. It can be challenging to setup and access a shared filesystem across all of the containers. The Kubernetes volume abstraction solves both of these problems. Familiarity with Pods is suggested.<br><br>https://kubernetes.io/docs/concepts/storage/volumes/ |

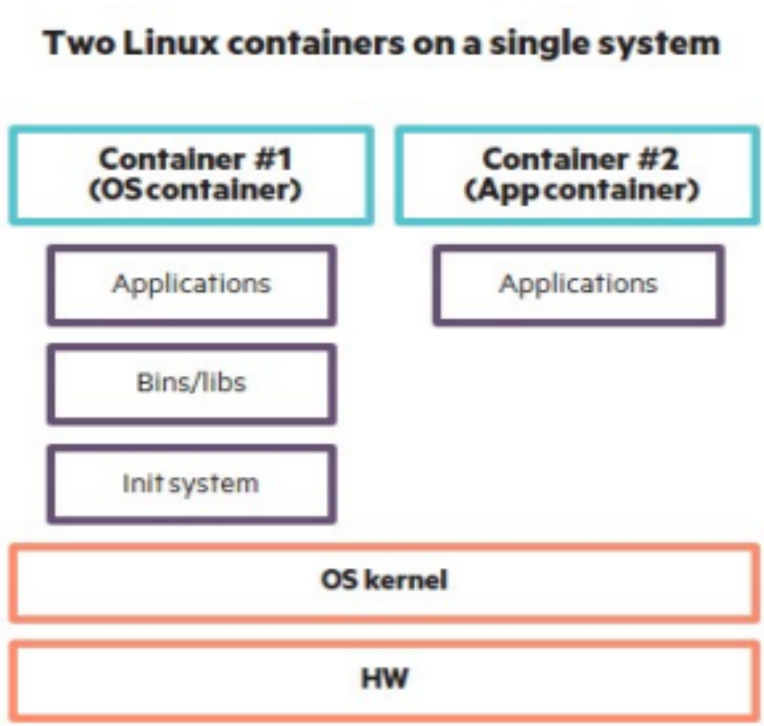
| Claim 1 | Accused Instrumentalities |
| --- | --- |
| | **Open Container Initiative**<br><br>**Image Format Specification**<br><br>This specification defines an OCI Image, consisting of an image manifest, an image index (optional), a set of filesystem layers, and a configuration.<br><br>The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run.<br><br>https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md |

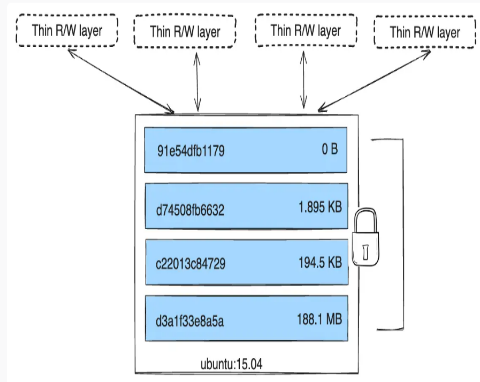
| Claim 1 | Accused Instrumentalities |
| --- | --- |
| | **Overview**<br><br>At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem layer changeset archives that will be unpacked to make up the final runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The image index is a higher-level manifest which points to a list of manifests and descriptors. Typically, these manifests may provide different implementations of the image, possibly varying by platform or other attributes.<br><br><br><br>https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md |

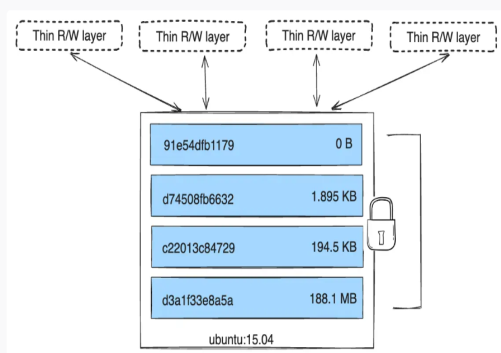
| Claim 1 | Accused Instrumentalities |
|---|---|
| | **OCI Image Configuration**<br><br>An OCI *Image* is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. This specification outlines the JSON format describing images for use with a container runtime and execution tool and its relationship to filesystem changesets, described in Layers.<br><br>This section defines the `application/vnd.oci.image.config.v1+json` media type.<br><br>https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md |

| Claim 1 | Accused Instrumentalities |
|---|---|
| | **Layer**<br><br>• Image filesystems are composed of *layers*.<br>• Each layer represents a set of filesystem changes in a tar-based layer format, recording files to be added, changed, or deleted relative to its parent layer.<br>• Layers do not have configuration metadata such as environment variables or default arguments - these are properties of the image as a whole rather than any particular layer.<br>• Using a layer-based or union filesystem such as AUFS, or by computing the diff from filesystem snapshots, the filesystem changeset can be used to present a series of image layers as if they were one cohesive filesystem.<br><br>**Image JSON**<br><br>• Each image has an associated JSON structure which describes some basic information about the image such as date created, author, as well as execution/runtime configuration like its entrypoint, default arguments, networking, and volumes.<br>• The JSON structure also references a cryptographic hash of each layer used by the image, and provides history information for those layers.<br>• This JSON is considered to be immutable, because changing it would change the computed ImageID.<br>• Changing it means creating a new derived image, instead of changing the existing image.<br><br>https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md<br><br>Containers only have access to resources that are defined in the image,<br>https://www.hpe.com/us/en/what-is/docker.html |

| Claim 1 | Accused Instrumentalities |
| --- | --- |
| | **DESCRIPTION** top<br>`The programs ld.so and ld-linux.so* find and load the shared objects (shared libraries) needed by a program, prepare the program to run, and then run it.`<br>https://man7.org/linux/man-pages/man8/ld.so.8.html |
| [1e] ii) wherein an instance of a SLCSE provided to at least a first of the plurality of software applications from the shared library is run in a context of said at least first of the plurality of software applications without being shared with other of the plurality of software applications and where at least a second of the plurality of software applications running under the operating system have use of a unique instance of a corresponding critical system element for performing same function, and | In each Accused Instrumentality, an instance of a SLCSE provided to at least a first of the plurality of software applications from the shared library is run in a context of said at least first of the plurality of software applications without being shared with other of the plurality of software applications and where at least a second of the plurality of software applications running under the operating system have use of a unique instance of a corresponding critical system element for performing same function.<br><br>When a Docker or Kubernetes image is used to create a container, it creates a separate and isolated instance of a runtime environment which is independent of other containers running on the same host. Each container has its own instance of base images and its own data. The containers run in isolation, ensuring that the SLCSEs stored in the shared library are accessible to the software applications running in their respective containers. The image includes essential system files, libraries, and dependencies required to run the software application within the container. The containers can share common dependencies and components using layered images. This means that multiple containers utilize the same base image to create an instance. When an instance of SLCSE is provided from the base image (i.e., from the shared library) to an individual container including application software, it operates in isolation and runs its own instance of the software application without sharing resources or critical system elements with other containers. This ensures that each container has its own isolated context. Docker or Kubernetes containers can share common dependencies and components using layered images. This means that multiple containers can utilize the same base image. Therefore, each container, containing the application software running under the operating system, utilizes a unique instance of the corresponding critical system element to execute the respective application software for performing a same function.<br><br>*See, e.g.*: |

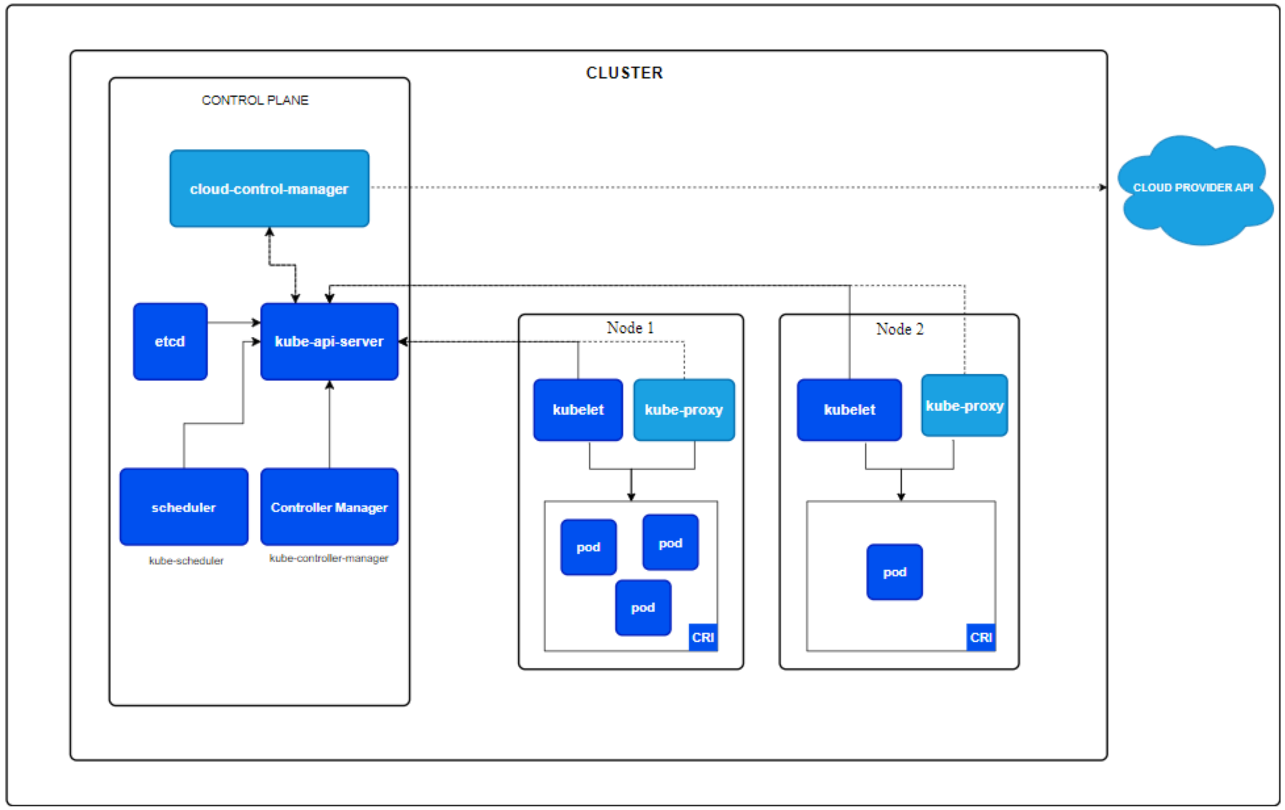
| Claim 1 | Accused Instrumentalities |
|---|---|
| | Containers provide the core runtime abstraction for the user applications. These containers provide isolation between user applications and the rest of the infrastructure. The containers are based on Docker.<br>https://support.hpe.com/hpesc/public/docDisplay?docId=a00097165en_us&docLocale=en_US&page=GUID-6B6676DB-AF5F-4555-B6AB-D2C11A89F320.html<br><br><br>https://h50146.www5.hpe.com/products/software/oe/linux/mainstream/support/whitepaper/pdfs/4AA6-2761ENW.pdf |

| Claim 1 | Accused Instrumentalities |
|---|---|
| | Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image.<br><br><br>https://docs.docker.com/storage/storagedriver/<br><br>Docker is used to create, run and deploy applications in containers. A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container.<br><br>https://www.techtarget.com/searchitoperations/definition/Docker-image |
| [1f] iii) wherein a SLCSE related to a predetermined function is provided to the first of the plurality of software applications for running a first instance of the SLCSE, and wherein a SLCSE for performing a same function is provided to the second of the plurality of software applications for running a second instance of the SLCSE simultaneously. | In each Accused Instrumentality, a SLCSE related to a predetermined function is provided to the first of the plurality of software applications for running a first instance of the SLCSE, and wherein a SLCSE for performing a same function is provided to the second of the plurality of software applications for running a second instance of the SLCSE simultaneously.<br><br>For example, in Docker or Kubernetes containers, each container operates independently, and a base image includes essential system files, libraries, and dependencies (i.e., SLCSEs) required to run the software application within the container. Based on information and belief, each element, such as system files, libraries, and dependencies (i.e., SLCSE) is associated with an execution of a predetermined function related to the application. When an image is used to create a container in the Accused Instrumentality, an instance of the SLCSE is provided to a software application. Therefore, |

| Claim 1 | Accused Instrumentalities |
| --- | --- |
| | different instances of the SLCSE are provided to different applications for performing a same function, simultaneously.<br><br>*See, e.g.*:<br><br>Docker is used to create, run and deploy applications in containers. A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container.<br><br>https://www.techtarget.com/searchitoperations/definition/Docker-image, Last accessed on June 14, 2023<br><br>A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.<br>https://docs.docker.com/get-started/overview/<br><br>Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image.<br><br>https://docs.docker.com/storage/storagedriver/ |

**Claim 2**

| Claim 2 | Accused Instrumentalities |
|---|---|
| 2. A computing system as defined in claim 1, wherein in operation, multiple instances of an SLCSE stored in the shared library run simultaneously within the operating system. | Each Accused Instrumentality comprises or constitutes a computing system as defined in claim 1, wherein in operation, multiple instances of an SLCSE stored in the shared library run simultaneously within the operating system.<br><br>For example, an individual host/node runs multiple containers and/or pods simultaneously, each of which has an instance of an SLCSE. When the multiple containers and/or pods run simultaneously, the multiple instances of the SLCSE stored in the shared library run simultaneously.<br><br>*See, e.g.*: |

| Claim 2 | Accused Instrumentalities |
|---|---|
| | <br>Kubernetes cluster architecture<br>https://kubernetes.io/docs/concepts/architecture/ |

| Claim 2 | Accused Instrumentalities |
|---|---|
| | **Containers**<br><br>Each container that you run is repeatable; the standardization from having dependencies included means that you get the same behavior wherever you run it.<br><br>Containers decouple applications from the underlying host infrastructure. This makes deployment easier in different cloud or OS environments.<br><br>Each node in a Kubernetes cluster runs the containers that form the Pods assigned to that node. Containers in a Pod are co-located and co-scheduled to run on the same node.<br><br>https://kubernetes.io/docs/concepts/containers/ |

<table>
<tr><th>Claim 2</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

# Kubernetes Scheduler

In Kubernetes, *scheduling* refers to making sure that Pods are matched to Nodes so that Kubelet can run them.

## Scheduling overview

A scheduler watches for newly created Pods that have no Node assigned. For every Pod that the scheduler discovers, the scheduler becomes responsible for finding the best Node for that Pod to run on. The scheduler reaches this placement decision taking into account the scheduling principles described below.

If you want to understand why Pods are placed onto a particular Node, or if you're planning to implement a custom scheduler yourself, this page will help you learn about scheduling.

https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/

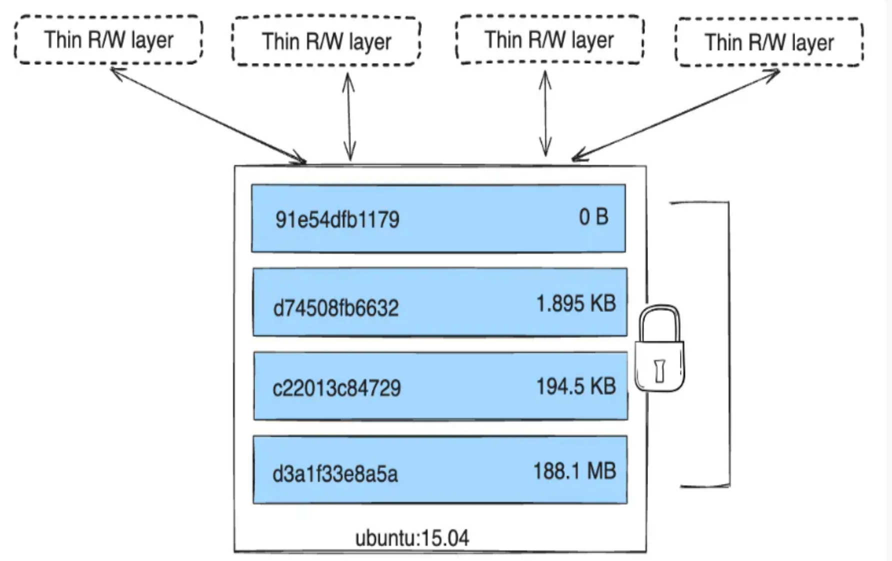
</td></tr>
</table>

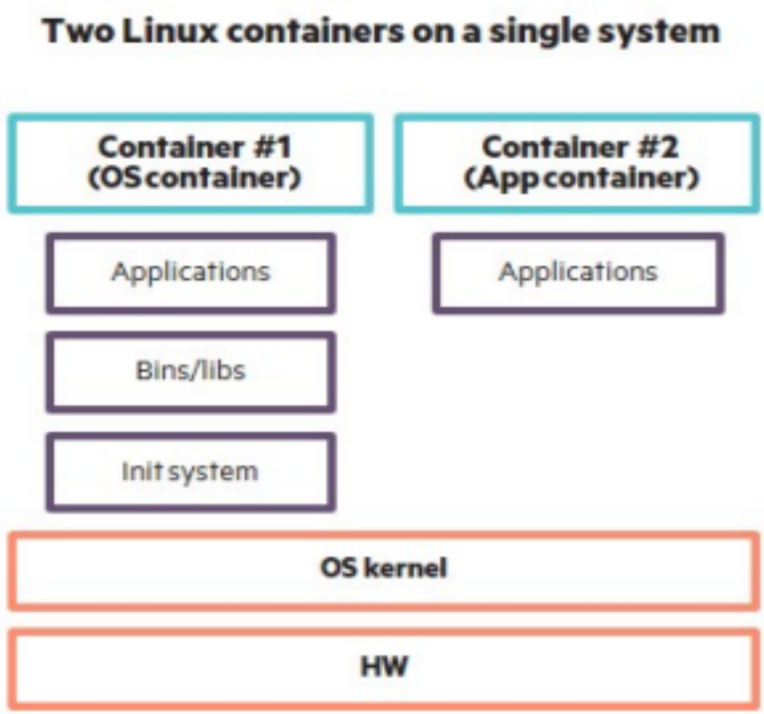
| Claim 2 | Accused Instrumentalities |
|---|---|
| | **Running containers**<br><br>Docker runs processes in isolated containers. A container is a process which runs on a host. The host may be local or remote. When you execute `docker run`, the container process that runs is isolated in that it has its own file system, its own networking, and its own isolated process tree separate from the host.<br><br>https://docs.docker.com/engine/reference/run/ |

**Claim 3**

| Claim 3 | Accused Instrumentalities |
|---|---|
| 3. A computing system according to claim 1 wherein OSCSEs corresponding to and capable of performing the same function as SLCSEs remain in the operating system kernel. | Each Accused Instrumentality comprises or constitutes a computing system according to claim 1 wherein OSCSEs corresponding to and capable of performing the same function as SLCSEs remain in the operating system kernel.<br><br>For example, both Docker and Kubernetes systems preserve the host kernel substantially unchanged; therefore the OSCSEs corresponding to the SLCSEs remain in the operating system kernel.<br><br>*See, e.g.*:<br><br>Containers provide the core runtime abstraction for the user applications. These containers provide isolation between user applications and the rest of the infrastructure. The containers are based on Docker.<br><br>https://support.hpe.com/hpesc/public/docDisplay?docId=a00097165en_us&docLocale=en_US&page=GUID-6B6676DB-AF5F-4555-B6AB-D2C11A89F320.html<br><br>The container starts with a base image, and the microservice is packaged into a container image and then deployed through the container platform. The container platform is based on<br><br>https://www.hpe.com/us/en/what-is/container-platform.html |

| Claim 3 | Accused Instrumentalities |
| --- | --- |
| | Container images<br><br>A container image is a ready-to-run software package containing everything needed to run an application: the code and any runtime it requires, application and system libraries, and default values for any essential settings.<br><br>https://kubernetes.io/docs/concepts/containers/<br><br>The idea of containerization is to isolate and package the application with all the dependencies in a container.<br><br>https://community.hpe.com/t5/hpe-blog-uk-ireland-middle-east/containerization-the-next-generation-of-virtualization/ba-p/7154442<br><br>Container image files are complete, static and executable versions of an application or service and differ from one technology to another. Docker images are made up of multiple layers, which start with a base image that includes all of the dependencies needed to execute code in a container. Each image has a readable/writable layer on top of static unchanging layers. Because each container has its own specific container layer that customizes that specific container, underlying image layers can be saved and reused in multiple containers. An Open Container Initiative (OCI)<br><br>https://www.techtarget.com/searchitoperations/definition/container-containerization-or-container-based-virtualization |

| Claim 3 | Accused Instrumentalities |
|---|---|
| | Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image.<br><br>https://docs.docker.com/storage/storagedriver/ |

| Claim 3 | Accused Instrumentalities |
|---|---|
| |   https://h50146.www5.hpe.com/products/software/oe/linux/mainstream/support/whitepaper/pdfs/4AA6-2761ENW.pdf |

**Claim 4**

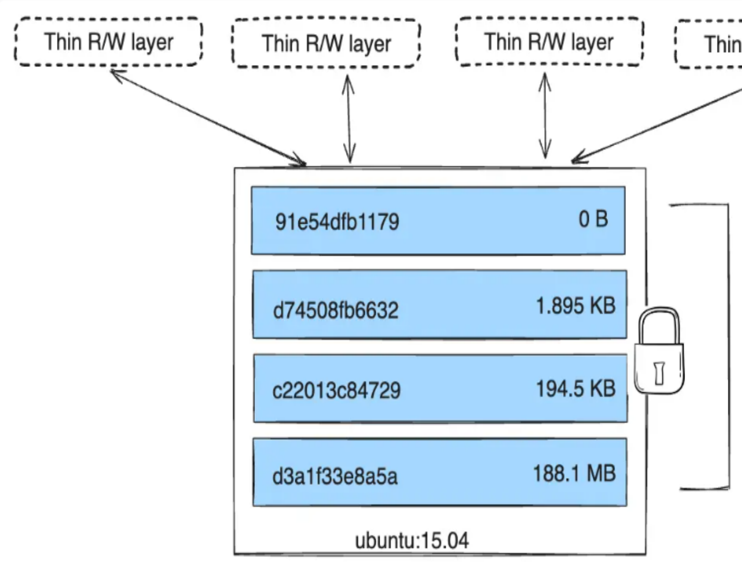
| Claim 4 | Accused Instrumentalities |
|---|---|
| 4. A computing system according to claim 1 wherein the one or more SLCSEs provided to one of the plurality of software applications having exclusive use thereof, use system calls to access services in the operating system kernel. | Each Accused Instrumentality comprises or constitutes a computing system according to claim 1 wherein the one or more SLCSEs provided to one of the plurality of software applications having exclusive use thereof, use system calls to access services in the operating system kernel.<br><br>For example, the SLCSEs in a container use system calls to access services in the operating system kernel. For example, the glibc library (or other similar library) in the container uses system calls to |

| Claim 4 | Accused Instrumentalities |
| --- | --- |
| | interface with the host Linux kernel. In general, system calls can be observed using a tool such as strace.<br><br>*See, e.g.*:<br><br>The **GNU C Library**, commonly known as **glibc**, is the GNU Project implementation of the C standard library. It is a wrapper around the system calls of the Linux kernel for application use. Despite its name, it now also directly supports C++ (and, indirectly, other programming languages). It was started in the 1980s by the Free Software Foundation (FSF) for the GNU operating system.<br><br>https://en.wikipedia.org/wiki/Glibc |

| Claim 4 | Accused Instrumentalities |
|---|---|
| | We can now get the process id directly from the cgroup. It will be located in the `cgroup.procs` file.<br><br>`### Terminal 2 - Worker Node ###`<br><br>`# Get the process id`<br>`$ cat cri-containerd-ceeeef06afe89c8223d33b11e8d9e0b207118ac4dac3af826687668ee1ee`<br>`16254`<br><br>`# Validate what is running under the process`<br>`$ ps aux \| grep 16254`<br>`azureus+    16254 0.0  0.1 713972 10476 ?       Ssl  15:04   0:00 ./faultyapp`<br>`azureus+   94806  0.0  0.0    7004  2168 pts/0    S+   16:22   0:00 grep --color=a`<br><br>Got it! With that, we can try to find out what is going out inside the app. Lets try to run `strace` to get some more insight.<br><br>`### Terminal 2 - Worker Node ###`<br><br>`$ sudo strace -p 16254 -f`<br>`...`<br>`# The app is trying to read a file port.txt`<br>`[pid 16269] openat(AT_FDCWD, "port.txt", O_RDONLY\|O_CLOEXEC <unfinished ...>`<br>`[pid 16254] epoll_pwait(5,  <unfinished ...>`<br>`# The file does not exist`<br>`[pid 16269] <... openat resumed>)       = -1 ENOENT (No such file or directory)`<br>`[pid 16254] <... epoll_pwait resumed>[], 128, 0, NULL, 0) = 0`<br>`[pid 16269] write(1, "Something went wrong...\\n", 24 <unfinished ...>`<br><br>After filtering the output, we can see the application is trying to read a text file called `port.txt`, and a few lines later, there is a message stating `ENOENT (No such file or directory)`. Let's create that file.<br>https://www.berops.com/blog/a-different-method-to-debug-kubernetes-pods |

**Claim 18**

| Claim 18 | Accused Instrumentalities |
|---|---|
| 18. A computer system as defined in claim 2 wherein SLCSEs are not copies of OSCSEs. | Each Accused Instrumentality comprises or constitutes a computer system as defined in claim 2 wherein SLCSEs are not copies of OSCSEs.<br><br>For example, in a typical case the SLCSEs come from a Linux distribution independent of the host operating system, and thus are not identical to the OSCSEs.<br><br>*See, e.g.*:<br><br>Hewlett Packard Enterprise provides publicly available base OS images for use in containerized clusters. These images extend the base OS images available from Docker hub by adding several packages that permit HPE Ezmeral Runtime Enterprise to manage container orchestration seamlessly and to improve the security of the container.<br>https://docs.ezmeral.hpe.com/runtime-enterprise/55/app-workbench-5-1/custom-base-images/AWB51_About_Custom_Base_Images.html<br><br>The idea of containerization is to isolate and package the application with all the dependencies in a container.<br>https://community.hpe.com/t5/hpe-blog-uk-ireland-middle-east/containerization-the-next-generation-of-virtualization/ba-p/7154442<br><br>Container images<br><br>A container image is a ready-to-run software package containing everything needed to run an application: the code and any runtime it requires, application and system libraries, and default values for any essential settings.<br><br>https://kubernetes.io/docs/concepts/containers/<br><br>Docker is used to create, run and deploy applications in containers. A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container. |

| Claim 18 | Accused Instrumentalities |
| --- | --- |
| | ttps://www.techtarget.com/searchitoperations/definition/Docker-image<br><br>Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image.<br><br><br>https://docs.docker.com/storage/storagedriver/<br><br>Containers only have access to resources that are defined in the image,<br>https://www.hpe.com/us/en/what-is/docker.html |